PARALLELIZATION OF GRAPH CONVOLUTIONAL NETWORK

Hareesh P. Nair
Department of Aerospace Engineering
Indian Institute of Technology Madras
Chennau, Tamil Nadu 600036
Email: ae20b027@smail.iitm.ac.in

ABSTRACT

The primary objective of the project is to parallelize the Graph Covolutional Network learning algorithm on large-scale graphs through efficient parallelizing methods, which include the usage of OpenMP. It introduces a novel approach by implementing the neural network architecture in C++, in contrast to the Python implementation, providing a low-level control and efficiency in memory management and faster execution. The parallelization of GCNs involves distributing message passing, feature aggregation, and graph convolutional layers across multiple threads. The parallelization of the program, along with some optimization, is expected to minimize computation time for different processes within the GCN algorithm, which includes message-passing, node updates, and gradient computations. The result of the project will be evaluated by comparing the time taken for training a dataset on the sequential and parallelized algorithm with different numbers of threads.

NOMENCLATURE

A Adjacency matrix
$H^{(l)}$ l^{th} hidden layer in the model
σ Activation function
W Weight matrix
D Diagonal matrix with entries D_{ii} as the degree of the i^{th} node
n_{in} Number of inputs
n_{out} Number of outputs

INTRODUCTION

Graphs arise in various real-world applications like social analysis, traffic prediction, and computer vision. They encode structural information to model entity relations, promising deeper insights. However, graph complexity hinders insights due to its non-Euclidean nature. To address this, embedding techniques learn graph features in lower dimensions. While effective, they often come with shallow learning mechanisms. Integrating deep learning with graph embedding results in the formation of Graph Neural Networks (GNNs), which can learn representations in much more detailed depth and hence, provide a higher accuracy on the data.

Over the recent years, graph neural networks (GNNs) have shown immense potential in recommendation systems, natural language processing, link prediction, etc. Among the GNNs, graph convoltional networks (GCNs) is the most broadly applied model.GCNs utilizes the information about a node and its locality to make predictions.

However, the scalability of these GNNs has been proven challenging. This certain challenge can be overcome by utilizing high performance computing methods to parallelize the background operations of GCN. This would involve simultaneous calculations, transformations and updates occuring at different parts of the algorithm providing a faster execution time to train a GCN model and hence providing scalability for the algorithm.

BACKGROUND

Graph convolution networks [1] utilizes a specific architecture design which allows it to process and analyze graph data efficiently. There several components present in this architecture which work together to learn a high level representation of nodes within the graph.

A graph is typically represented as an adjacency matrix or a node feature matrix. The adjacency matrix is an $N \times N$ matrix in which the element $A_{ij} = 1$ for if the i^{th} node is connected to j^{th}. For an un-directed graph, the adjacency matrix will always be symmetric. In a node feature matrix, each node in the graph can have associated features. These features are often represented

as a feature matrix, where each node corresponds to a node and each column corresponds to a feature dimension.

Graph Convolution Operation

$$H^{(l+1)} = \sigma(AH^{(l)}W^{(l)}) \tag{1}$$

The features of the $(l+1)$ layer is obtained by passing the output of the previous layer through the activation function much like any dense neural network layer or a normal convolution layer. The difference between a convolutional neural network and GCN is that, each vertex of the graph will not have the same number of neighbours. This process can also be considered as the forward propogation of the graph.

Message Passing/ Neighbour aggregation

By passing just the adjacency matrix onto the GCN layer, we will not gain information on how the vertex is affected by its neighbours. Hence we apply a transformation to the adjacency matrix to account for the above factors.

Self-connection : Each vertex in the graph is given a connection to itself

$$\tilde{A} = A + I \tag{2}$$

Normalization : As the adjacency matrix gets multiplied with with the weights and feature representations, the scale of features and weights will change dramatically [2]. In order to avoid this, we normalize the adjacency matrix:

$$\hat{A} = D^{-\frac{1}{2}}\tilde{A}D^{-\frac{1}{2}} \tag{3}$$

Feature aggregation : For each node, its feature is updated by taking a weighted average of the features of its neighbours and itself. The weights are determined by the normalized adjacency matrix and the learnable weight matrix within the convolution layer [3].

Finally we obtain an equation for the GCN model as

$$H^{(l+1)} = \sigma(\hat{A}H^{(l)}W^{(l)}) = \sigma(D^{-\frac{1}{2}}\tilde{A}D^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{4}$$

METHODOLOGY

The model architecture

Input layer : The model architecture developed in C++ comprises of multiple GCN layers. The inputs given to the model are : adjacency matrix, labels of the nodes, number of hidden layers, size of each hidden layer, and the activation layer.

Intialization of weights : The weights are initialized by a modified Glorot and Benigo [4] method of initialization which uses provides random values based on the probability distribution of a zero-mean Gaussian distribution with the standard deviation equal to $\frac{20}{n_{in}+n_{out}}$.

Output layer : The output layer is a softmax function which gives the probabilities of the node for each label classes.

Optimization : The backward propogation is carried out with the pre-defined parameters, the learning rate and weight decay factor for the L2 regularization, which updates the parameters for the specified number of epochs. The loss is calculated through the cross-entropy for each epochs.

The output of the model is the list of embeddings for the graph nodes obtained through the training, the accuracy of the model, the train loss and the test loss for each epoch.

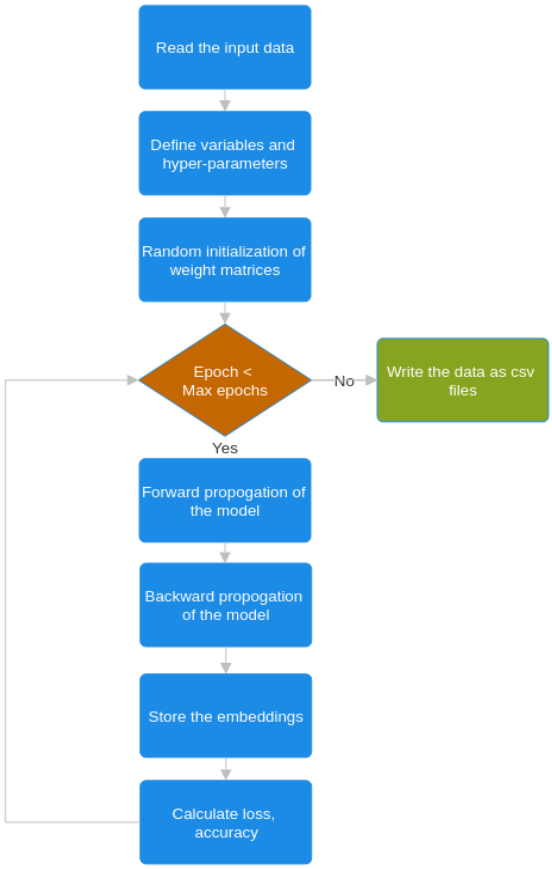


FIGURE 1: Model workflow

Data

Zachary's karate club network is a classic example in network analysis, often used to demonstrate community detection algorithms. It represents the social interactions within a university karate club, where nodes represent club members, and edges between nodes represent friendships or interactions outside the club.To identify community labels for each node in the karate club network, the code utilizes the greedy modularity maximization algorithm (greedy_modularity_communities) from NetworkX available in python. This algorithm partitions the nodes into communities based on maximizing the modularity score, which measures the strength of division of a network into communities compared to random partitions.

Number of nodes : 34
Number of edges : 78

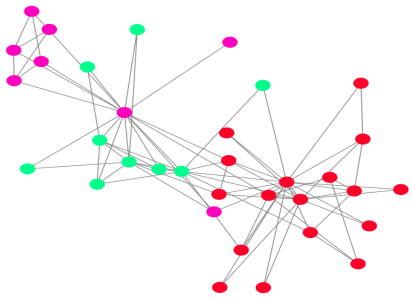


FIGURE 2: Network of Zachary's karate club

Parallelization framework

The algorithm is parallelized using OpemMP in C++. The parallelization occurs mainly in two layers.

Matrix calculation level : The base code contains number of operations on matrices which utilizes OpenMP multithreaded parallelization on loops. This include matrix product, matrix sum, transpose, dot product, norm calculation, reshaping, flattening, etc. The clauses used include but not limited to : omp parallel , for, reduction, atomic, collapse etc.

Parameter updation level : Several steps in the training involves updating the weight matrices, initializing temporary matrices for calculation of the updates etc. which all have been parallelized across multiple threads. The parallelization of this level occurs in forward propagation and backward propagation which itself utilizes the matrix operations. Some of the updation and calculation includes : calculation of activation functions of matrices, weight and bias updates etc.

The training loop of the model is inherently serial because of the data dependency of the layers. However, the sub-processes inside the training loop efficiently utilizes multiple threads in order to accelerate the calculation process.

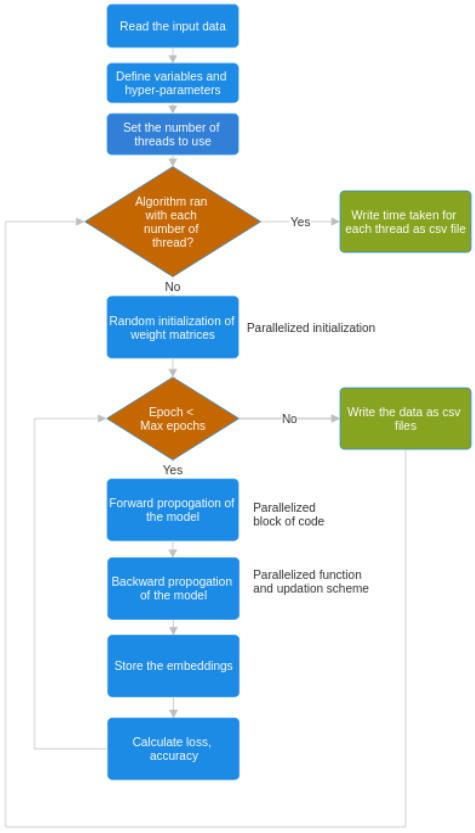


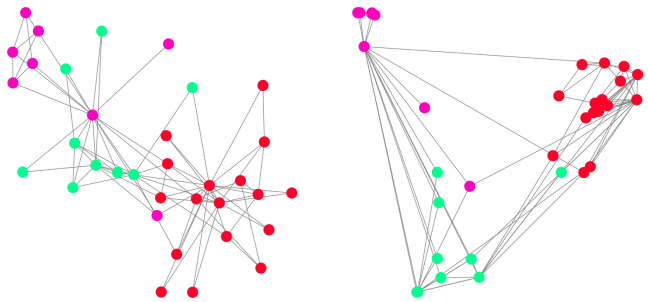
FIGURE 3: Parallel moodel workflow

RESULTS

Serial Code

The serial code was run and timed and the following results were obtained.

Network Analysis



(a) Graph embedding after training (b) Graph embedding after training

FIGURE 4: Graph embedding on serial code

Model parameters

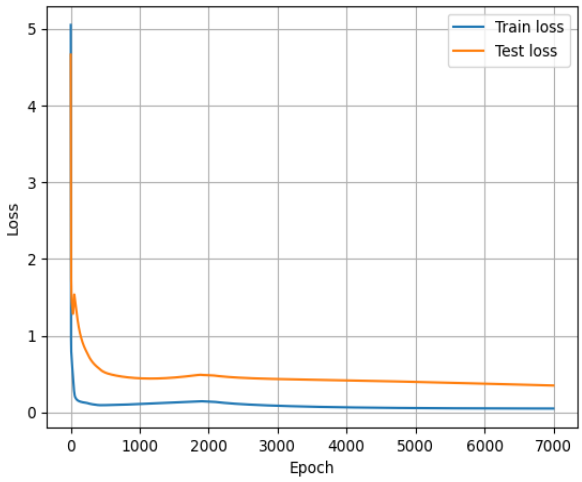


FIGURE 5: Loss vs epochs

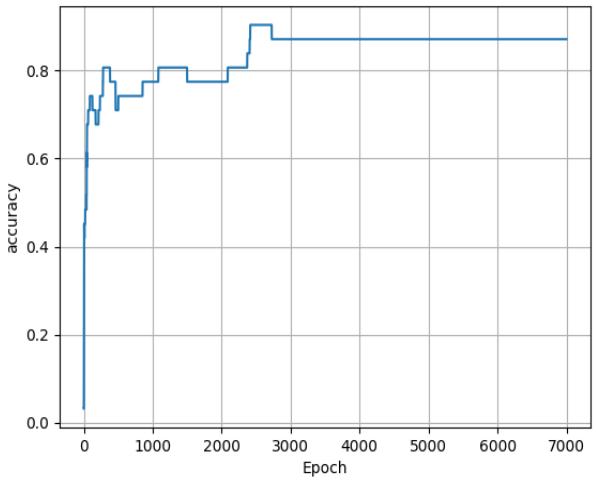
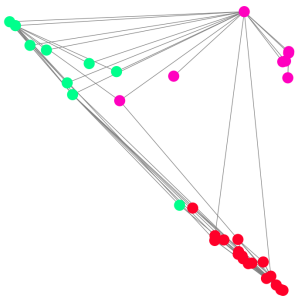


FIGURE 6: Accuracy vs epochs

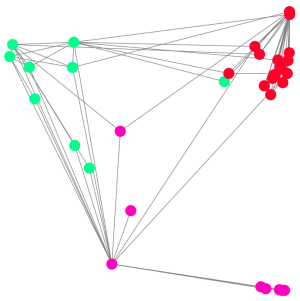
Parallel code

The parallel code developed in C++ using OpenMP was tested with several number of threads. The following results were obtained

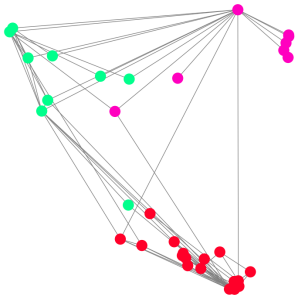
Network Analysis



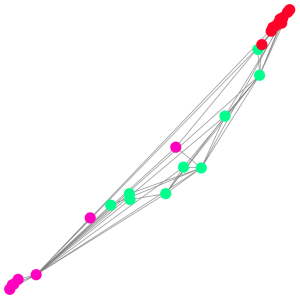
(a) Graph embedding for $p = 2$



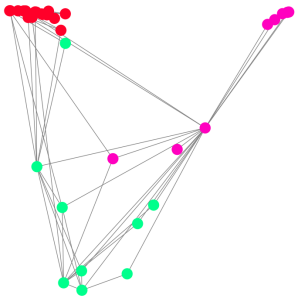
(b) Graph embedding for $p = 4$



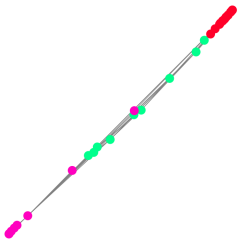
(c) Graph embedding $p = 6$



(d) Graph embedding for $p = 8$

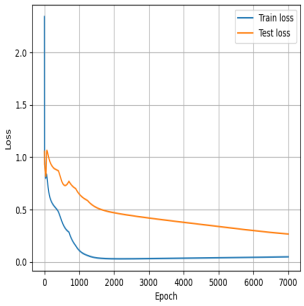


(e) Graph embedding for $p = 10$

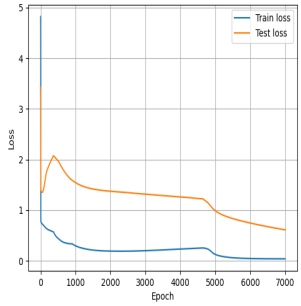


(f) Graph embedding for $p = 12$

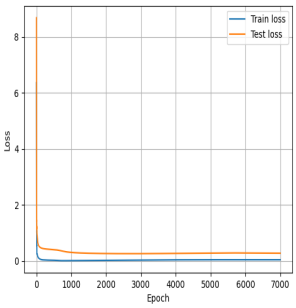
FIGURE 7: Graph embedding on parallel code



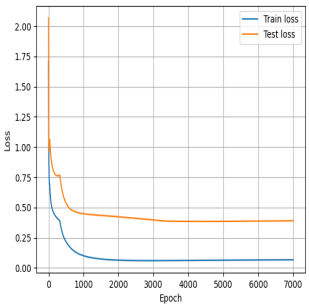
(a) Loss vs epochs for $p = 2$



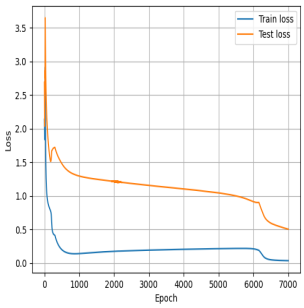
(b) Loss vs epochs for $p = 4$



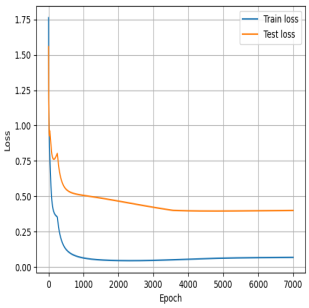
(c) Loss vs epochs for $p = 6$



(d) Loss vs epochs for $p = 8$



(e) Loss vs epochs for $p = 10$



(f) Loss vs epochs for $p = 12$

FIGURE 8: Graph embedding on parallel code

Model Parameters

Performance evaluation

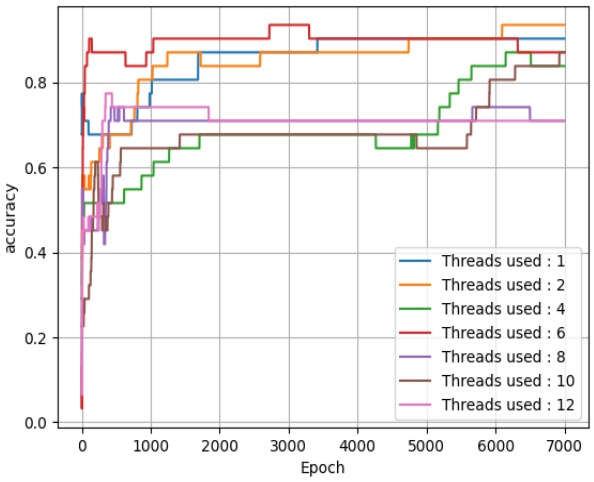


FIGURE 9: Accuracy vs epochs for the parallel code

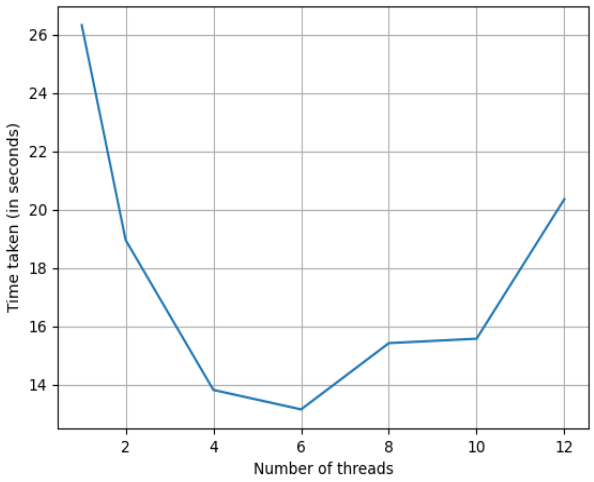


FIGURE 10: Time taken vs number of threads

OBSERVATION

Serial code

It can be seen that the serial program for GCN developed in C++ runs well and an accuracy of about 90% can be achieved.

From Fig. 4 we can say that the program was successful in clustering the nodes according to their labels and the labels with most connectivity to others is also embeded close to them compared to the ones the that have less connections.

The serial code takes about **30 seconds to run 7000 epochs**. The exact value can be obtained by running the code which shall be attached with the submission.

Parallel code

From Fig. 7, the network analysis of the parallel code is not consistent across all the threads but in all the cases, the algorithm is able to cluster most of the nodes according to their labels.

The loss function does not converge to the same value for most of the cases. For some, the test loss converges around 0.4 and for other it converges around 0.2 as seen in Fig. 8.

As displayed in Fig. 9, the accuracy for different threads is different but all of them lie in the range of 90% to 70%.

Finally from Fig. 10, we can see that the execution time of the code kept decreasing as the number of threads increased from 1 to 6. For thread numbers more than 6, the execution time kept increasing.

INFERENCE

The weight initialization part of the code, using Glorot and Benigo method utilized assigning random values based on a zero-mean Gaussian distribution function with the standard deviation of value depending on size on input and output. Due to the random initialization, the algorithm cannot have the same starting point on the loss curve each time it runs. Due to that fact, the optimal minimal point may not be achieved each time the code is executed.

Since the code does not execute with the same starting point each time it runs, the path it takes to reach the minimum also varies. Hence the algorithm does not reach the optimal minimum point as it iterates through different number of threads for execution, resulting in different loss curves, accuracies and hence the different network embedding position, of which results in some skewed shape.

Since the algorithm works faster and provides similar accuracies, for higher number of threads, it can be inferred that the parallelization of the code works smoothly as it reduces time of execution by more than 50% using 6 threads.

The reason for the higher execution time for the algorithm using more than 6 threads may be due to the overhead of initializing more threads for the process. Hence if we were

to scale up the program by increasing the number of nodes, we may see an improvement in the running time even after using more than 6 threads.

CONCLUSION AND FUTURE WORK

The parallelization of the GCN algorithm using OpenMP seems to successful and it shows the potential of training GCN much faster than a serial code. The accuracy of the model in the parallelized version is consistent with the accuracy of the serial code and hence proves that GCN algorithm is scalable with a suitable parallelization scheme.

The parallelization algorithm can also hope to see some improvement in the optimization method as the accuracies obtained in each of the training is not consistent. The code can also be parallelized using GPU accelerator or a hybrid GPU-CPU parallelization which may provide a much more faster execution time.

ACKNOWLEDGMENT

I would like to thank Prof. Kameshwara Rao for offering the course ID5130 - Parallel Scientific Computing in IIT Madras, which has helped me learn the basics of the OpenMP which I have extensively used in the project. I would also like to thank Zak Jost, the creator of the website and youtube channel AI Overloads, which had helped me cover the basics of Graph Neural Networks and hence providing me with the motivation to do this project. I would also like to thank the creators of LaTeX, which helped me document my work in such an organized manner.

REFERENCES

[1] Zhang, S., Tong, H., Xu, J., and Maciejewski, R., 2019. "Graph convolutional networks: a comprehensive review". *Computational Social Networks,* ***6***(1), Nov.

[2] Arcidiacono, S., 2021. What makes graph convolutional networks work?, May.

[3] Jepsen, T. S., 2019. How to do deep learning on graphs with graph convolutional networks, May.

[4] Kelesis, D., Fotakis, D., and Paliouras, G., 2024. Informed weight initialization of graph neural networks and its effect on oversmoothing.